First Hit Fwd Refs

End of Result Set

Generate Collection Print

L21: Entry 1 of 1 File: USPT Nov 17, 1998

DOCUMENT-IDENTIFIER: US 5838968 A
TITLE: System and method for dynamic resource management across tasks in real-time operating systems

Brief Summary Text (6):
This application is related to the U.S. patent application Ser. No. 08/541,565 entitled: "MULTI-MEDIA ENGINE INCLUDING PROCESSOR USING VARIABLE LENGTH INSTRUCTIONS" to James T. Battle, Andy C. Hung, and Stephen C. Purcell filed Nov. 10, 1995, and the U.S. patent application Ser. No. 08/556,416 entitled "A SYSTEM AND METHOD FOR FAST CONTEXT SWITCHING BETWEEN TASKS" to Denis Gulsen filed Nov. 10, 1995 the contents of which are hereby incorporated by reference in their entirety.

Brief Summary Text (9):
An example of a physical constraint is limited physical memory. Physical memory is also a constraint in non-realtime multi-tasking environments. Virtual memory is an example of a solution to alleviate physical memory limitations. The technique of virtual memory allows application programs to believe they have access to physical memory beyond what is actually available in hardware. This is accomplished through various techniques such as swapping and paging which relocate physical memory pages to backing store without the knowledge of the application. Traditional virtual memory techniques are not currently usable in real-time systems because the latency associated with copying physical memory pages to backing store is too long and time delays are indeterminate.

Brief Summary Text (10):
In order for a real time task to meet its schedule or processing deadline, it must have access to the critical system resources necessary for processing. Previous systems attempt to manage resource constraints by simply checking once, usually at task initiation, whether there are enough resources available for the task to execute. Such systems may also provide rudimentary static allocation through a reservation system. In a reservation system, a task requests the resources it needs from the system, sometimes at task invocation, other times at system initialization. The system withholds those resources from the available pool for future use by the task. With static allocation, or reservation, of resources, resources can become fragmented or under-utilized easily. Additionally, the system cannot readjust resources to a particular task once it has been allocated, or the other tasks may become resource-starved. Thus systems in which task needs or resource utilization may fluctuate, either because tasks' have terminated or because the tasks needs have changed, cannot readjust the resources available to other tasks. Static system resource optimization is thus limited to manual methods which can only approximate actual run-time optimal usage.

Brief Summary Text (13):
In order to understand the uses of dynamic resource allocation a brief discussion of real time operating system programming is in order. Real time systems are characterized by limited resources (hardware) and time criticality of operations. Real time systems are often embedded systems where the system cannot be readjusted or restarted easily. For example control processors in manufacturing environments,

real time medical monitoring systems, or remote satellite imaging systems. Programmers of these systems are faced with resource limitations (hardware in satellites can't change), or time criticality constraints (patients may die if the monitoring software misses a hardware interrupt giving the system vital input). The unavailability of necessary resources, such as physical memory, main processor, or input/output bus cycles can cause problems in these environments. Therefore programmers must expend precious instruction cycles to check and verify that the resources they need are available or can be obtained.

Brief Summary Text (15):
Finally programmers must manually determine how to balance resources between high priority tasks and low priority tasks. If a low priority task requires a large number of resources, but is not time critical, programmers have little choice but to hope that those resources are available when the low priority task runs. In static reservation systems a low priority task cannot pre-allocate all the resources it needs since that would make those resources unavailable for time critical tasks. Yet if a time critical task fails to deallocate its resources, the low priority task may never be able to execute since it cannot acquire the resources necessary to run.

Brief Summary Text (17):
The present invention provides a new method for optimal resource management between tasks in a real time multitasking environment. It is one aspect of the present invention to provide a resource allocation mechanism for multiple resources. It is another aspect of the invention to provide for dynamic resource management facilitating migration of resources from one task to another and from resource to resource. It is a further aspect of this invention that the allocation of resources is globally optimized and dynamically managed across all tasks in the real time operating system.

Detailed Description Text (20):
Register file 30, is, e.g., a static RAM, and includes a plurality of individually addressable units. Each addressable unit is, e.g., a 72 bit Dword. Software and hardware are coupled to register file 30 for, e.g., instruction caching and storing instruction operand and result data. Register file 30 provides a buffer for other hardware functions such as peripheral bus control 90, emulation register control, and direct device access 80 and is coupled to each of peripheral bus control 90, and direct device access 80. The buffer of register file 30 can be used in a variety of programmatic ways such as first in/3first out (FIFO) queue, random access, or accessed via memory mapping or other common access techniques.

Detailed Description Text (23):
Real time tasks execute under XOS 180 on the media engine chip, while resource manager 170 is responsible for creating and dynamically managing the resources available to tasks. Tasks have a task structure commonly known in the art as a task control block that contains the control information necessary for a task to run, be suspended and resumed.

Detailed Description Text (32):
Referring to FIG. 3, task 350 has 3 task resource utilization records, 310, 320, and 330. Record 310 contains a list of resources that task 350 would like to have available while executing. If task 350 cannot be allocated the resources specified in 310, task 350 could execute and perform its functions with the resources specified in record 320 or 330. However, if the resources specified in record 330 are not available, then task 350 could not execute and perform its functions properly.

Detailed Description Text (36):
If the media engine subsystem becomes resource constrained, and tasks have difficulty gaining access to needed resources then resource manager 170 must decide

whether to lower the available resources for current tasks, or fail the task allocation request. The act of retrieving resources from an existing task is called degradation. Degradation occurs when a task is asked to give up some of its resources and move to a lower run level.

Detailed Description Text (43):
A newly allocated task requires more resources than are currently available

Detailed Description Text (52):
In step 510, the process first checks the currently available amount of resources. The routine GetPlatformUsage provides the necessary information.

Detailed Description Text (53):
If the requested resources are available, as testing in step 520, then the request is fulfilled and the resources are allocated in step 530. If the resources are not available the resource manager 170, executing step 540, will scan through task list 400 and perform two functions.

Detailed Description Text (65):
In step 560, the resources recovered are compared against the deficit. If there are now enough resources available, QueryResult tells the tasks to set themselves to the specified run levels computed in the optimal system utilization level, step 580, and the process executes step 530 to allocation the resources. Otherwise in step 570, the request for additional resources is denied.

Detailed Description Text (66):
In addition to degrading tasks, resource manager 170 seeks to increase resources available to tasks by promoting tasks.

Detailed Description Text (70):
Resources become available due to task termination or task freeing resources.

Detailed Description Text (72):
In step 610, the process first checks the currently available amount of resources. The routine GetPlatformUsage provides the necessary information.

Detailed Description Text (73):
If the requested resources are available, as testing in step 620, then the request is fulfilled and the resources are allocated in step 630. If the resources are not available the resource manager 170, executing step 640, will scan through task list 400 and perform two functions.

Detailed Description Text (84):
In step 660, the resources recovered are compared against the deficit. If there are now enough resources available, QueryResult tells the tasks to set themselves to the specified run levels computed in the optimal system utilization level, step 680, and the process executes step 630 to allocation the resources. Otherwise in step 670, the request for additional resources is denied.

Detailed Description Paragraph Table (3):
QueryResult handler(DWORD ref, RMAdviseCommand command, RMUsageInfo *usageInfo, const RMUsageInfo& delta)
Effects: Implements commands defined in command structure including degradation usageInfo pointer to an RMUsageInfo block delta: for degradation, the values in this resource vector represent the current resource deficit/resources needed. For promotion, the values represents current free resources available. (Note: in the case of degradation, a resource with a negative deficit can be ignored - there is no conflct for that resource . In the case of promotion, no resource value should be set negative

CLAIMS:

1. A method of dynamic resource management on a data processing system including a processor, at least one system resource, and a plurality of tasks, the processor being coupled to the system resource and capable of executing the plurality of tasks, the method comprising the steps of:

executing instructions on the processor to create at least two tasks capable of executing on the processor, each of the at least two tasks having a priority;

creating at least one task resource utilization vector for each of the at least two tasks, the task resource utilization vector comprising the quantity of the at least one system resource that each of the at least two tasks prefers to utilize while executing on the processor; and

dynamically varying the quantity of the at least one system resource that the at least two tasks have allocated based on the availability of the at least one system resource and the priorities of the at least two tasks.

4. A method of dynamic resource management on a data processing system including a processor, a plurality of system resources, and a plurality of tasks, the processor being coupled to the system resource and capable of supporting the tasks, the method comprising the steps of:

executing instructions on the processor to create a plurality of tasks capable of executing on the processor, each of the plurality having a priority;

creating a plurality of task resource utilization records for each of the plurality of tasks, each task resource utilization record of the plurality comprising quantities of the pluralities of system resources that each of the plurality of tasks qualitatively prefers to utilize while executing on the processor;

for each task resource utilization record of the plurality, assigning a run level to the task utilization record reflecting the associated task's ability to perform its work when allocated the resources according to each task resource utilization record; and

dynamically varying the quantity of the plurality of system resources that the plurality of tasks have allocated based on the availability of the plurality of system resources and the priorities of the plurality of tasks.

6. A system for dynamic resource management on a data processing system including a plurality of processors, a plurality of system resources, and a plurality of tasks, the plurality of processors being coupled to the plurality of system resources and capable of supporting the tasks, the system comprising:

instructions that when executed on at least one of the plurality of processors create a plurality of tasks capable of execution on at least one of the plurality of processors, each of the plurality of tasks having a priority;

means for creating a plurality of task resource utilization records for each of the plurality of tasks, each task resource utilization record of the plurality comprising quantities of the pluralities of system resources that each of the plurality of tasks qualitatively prefers to utilize while executing on the at least one processor;

means for assigning a run level to each of the task utilization records of the plurality of tasks, each run level reflecting the associated task's ability to perform its work when allocated the resources according to each task resource utilization record; and

means for dynamically varying the quantity of the plurality of system resources that the plurality of tasks have allocated based on the availability of the plurality of system resources and the priorities of the plurality of tasks.

7. A method of dynamic resource management on a data processing system including a processor and a system resource coupled to the processor, the method comprising the steps of:

executing instructions on the processor to create a plurality of tasks capable of execution on the processor, each of the tasks having a priority;

creating a task resource utilization vector for each of the plurality of tasks, the task resource utilization vector including at least one task utilization resource record specifying a resource quantity request; and

dynamically varying the resource quantity requests for each of the plurality of tasks based on the availability of the system resource and the priorities of at least two of the plurality of tasks.

14. A method of dynamic resource management on a data processing system including a plurality of processors and a system resource coupled to the plurality of processors, the method comprising the steps of:

executing instructions on at least one of the plurality of processors to create a plurality of tasks capable of execution on any of the plurality of processors, each of the tasks having a priority;

creating a task resource utilization vector for each of the plurality of tasks, the task resource utilization vector including at least one task utilization resource record specifying a resource quantity request; and

dynamically varying the resource quantity requests for each of the plurality of tasks based on the availability of the system resource and the priorities of at least two of the plurality of tasks.

First Hit Fwd Refs

End of Result Set

☐ Generate Collection Print

L19: Entry 1 of 1 File: USPT Jul 8, 2003

DOCUMENT-IDENTIFIER: US 6591358 B2
TITLE: Computer system with operating system functions distributed among plural microcontrollers for managing device resources and CPU

Abstract Text (1):
A hardware/firmware layer comprising a Device Manager, an Information Manager, a Memory Manager, and a Process Manager contained in one or more semiconductor chips is disclosed. The hardware/firmware layer eliminates the need for an operating system. Each of the Managers comprises a microcontroller associated with a firmware embedded in ROM or Flash memory that contains instruction sets that cause the microcontroller to provide a designated task of device management, information management, memory management and process management. In another aspect of the invention, devices connected to the computer system are "smart devices," each device having a device microcontroller and embedded device drivers in a ROM or Flash memory. The hardware/firmware of the present invention does not need to search for available devices, provide diagnostic tests or obtain device drivers to communicate with the devices. Instead, the device microcontroller uses the embedded device driver to perform configuration and self diagnostic test as well as device operations. If the device is operational, the device microcontroller sends an identification signal to the hardware/firmware layer of the present to indicate availability of the device.

Brief Summary Text (5):
FIG. 1 illustrates a conventional computer system architecture 100 comprising a hardware platform layer 200, a firmware layer 300, an operating system layer 400 and an application programs layer 102. The hardware platform layer 200 is the physical layer of the computer system that performs the actual operations of the computer system. The firmware layer 300 performs, among others, the interface between the hardware platform layer 200 and the operating system layer 400. The operating system layer 400 is a software layer that performs the management of the computer resources such as processor resource management, memory allocation management, device resources management, and data file management. The operating system is also the base upon which application programs are built. The application programs layer 102 comprises computer programs that provide instruction sets that manipulate and/or process data in accordance with a desired result. Examples are word processor, database, spread sheet and web browser programs.

Brief Summary Text (10):
Referring now to FIG. 3, according to one power-up sequence, at stage 302, the computer is powered on or a reset signal is received in which the computer forces the components within the computer including accessible devices to a reset logic state. At reset logic state, the computer does not "know" its actual configuration including what devices including those on the expansion cards are attached to the computer. After a predefined period of time has passed in which the power supply has stabilized, at stage 304, the CPU starts at a starter address that points to the ROM in which the BIOS is located. The POST routine of the BIOS is initiated and it tests the dynamic random access memories (DRAMS) that make up the main memory along with certain devices and components of the system to determine their

operability. During the testing process, a copy of the BIOS is retrieved from the ROM and is shadowed into the main memory. The BIOS has a set of instruction routines that prepares the computer system to receive the operating system from an initial load device (IPL) which may be a disk drive. At stage 306, the BIOS attempts turn off all the devices to determine which devices (i.e. IPLs) may be used to find and launch the operating system. IPLs are detected at this stage because IPLs cannot be turned off. At stage 308, the BIOS turns on the devices and places non-IPLS in wait state to be initialized by the operating system. At stage 310, the BIOS executes a bootstrap routine that causes the kernel of the operating system (usually contained in the hard-disk drive) to load into the main memory. At this stage, the hardware control by the firmware (i.e., the BIOS) is passed to the software (which is the kernel). The kernel 402 provides the core function of the operating system which is computer resource management such as process execution, memory management, dynamic linked library management, scheduling, file system management, I/O services and user interface presentation, among others.

Brief Summary Text (11):
At stage 312, the kernel initiates an isolation procedure that isolates the devices individually. The key to the isolation protocol is that each device contains a unique 72-bit number known as a serial identifier. Once a device is isolated, it is assigned a Card Serial number (CSN) that is unique to the assigned device and serves as a "handle" in which the operating system identifies the device and with which the device identifies itself, for instance when generating an interrupt. At stage 314, the kernel reads the isolated devices individually for resource requirements of the device. The resources required by the devices include DMA, interrupt request (IRQs), I/O and memory addresses. At stage 316, the kernel creates a comprehensive list of the resource requirements of each device. At stage 318, because the kernel knows the available system resources, the kernel allocates the available resources to the devices as needed while ensuring the resource allocation is non-conflicting. At stage 320, as the kernel allocates system resources to the devices an allocation map is created and stored in memory. In addition to the creation of the allocation map, At stage 322, using the identification number provided by the device, the kernel identifies the associated device driver that is usually stored in the hard-disk drive. Should the device driver be unavailable, the kernel will prompt the user to provide the device driver. All device drivers associated with the detected devices are loaded into the main memory which is used by the kernel to control the devices. Further details of the power-up sequences may be found in Plug and Play ISA Specification. The isolation, interrogation of the various devices and loading of the device drivers into the main memory is time consuming and also reduces the available main memory.

Brief Summary Text (13):
The operating system also organizes the instructions from application programs into chunks called threads. A thread can be thought of as a packet of instructions that can be "chewed" for execution by the CPU. The operating system breaks the operation of multiple application programs into threads for sequential execution thus allowing the CPU to simultaneously support several application programs known as multi-tasking. Multi-tasking in one example increases the speed of a computers operation by allowing various devices to operate without idling the CPU. Usually, the CPU executes instructions much more quickly than data can be read and written into a storage device. Thus, the CPU would be idle if it had to wait for data to be written or read from a storage device. The use of threads allow the operating system to reassign the CPU whenever a task must be performed for a slow component of the system. For example, the processing of instructions from a first application program may be suspended whenever data must be read from a disk drive. The CPU may then execute a thread from another application program while the data is being read, and resume processing of the instructions from the first application program, once the data has been read.

Brief Summary Text (14):

The computer using a bus architecture usually has one bus in which the CPU and the various devices communicate through. Thus, the operating system controls a flow of instructions to the CPU from application programs, and temporal suspension of CPU processing of application program instructions to allow for the I/O communication by various devices such as the data transfer from the disk drive to the main memory via the DMA controller.

Brief Summary Text (15):
The computer system using an operating system described above has an undesirable lengthy power-up sequence that inconveniences the user and consumes valuable memory space. A known method uses a faster CPU to speedup the power-up sequence. However, a faster CPU is expensive and increases the cost of the computer. The computer system relying on the operating system is subject to "crashes" perhaps due to a tainted application program it had executed, or due to errors resulting from handling numerous interrupts and call procedures during multi-tasking. In addition, the operating system is subject to virus attacks that may render the computer system inoperational as well as destroying valuable data files. What is needed is a computer system and method that solves these and other shortcomings.

Brief Summary Text (18):
In one general aspect, a computer system comprises a central processing unit (CPU), a main memory, a further unit that includes a first microcontroller and a first memory containing a first set of instructions configured to cause the microcontroller to manage CPU operations, and a plurality of trace links connecting the further unit to the CPU and the main memory to facilitate communication between the further unit, the CPU and the main memory. Other features include at least one device, a trace link connecting the device to the further unit, and the further unit further includes a second microcontroller and a second memory containing a second set of instructions configured to cause the second microcontroller to manage the device; the further unit further includes a third microcontroller and a third memory containing a third set of instructions configured to cause the third microcontroller to manage memory operations; the further unit further includes a fourth controller and a fourth memory containing a fourth set of instructions configured to cause the fourth controller to manage data operations; the plurality of microcontrollers in the further unit are connected together to communicate with each other; the further unit includes a cross-bar switch to connect the plurality of microcontrollers; the device includes a fifth microcontroller and a fifth memory containing a fifth set of instructions configured to cause the fifth microcontroller to control the device operations, the fifth microcontroller in communication with at least the second microcontroller; the fifth set of instructions in the fifth memory further configured to cause the fifth microcontroller to test the device and if the device is operational the fifth set of instructions is configured to cause the fifth microcontroller to signal at least the second microcontroller to indicate availability of the device; the fifth set of instructions in the fifth memory further configured to cause the fifth microcontroller to signal at least the second microcontroller to indicate availability of the device includes sending device identification and required resource data; the second set of instructions in the second memory further configured to cause the second microcontroller to receive the signal indicating availability of the device and allocating available resources to the device.

Brief Summary Text (19):
In another aspect of the invention an apparatus for managing computer operations comprises a first microcontroller and a first memory containing a first set of instructions configured to cause the microcontroller to manage central processing unit (CPU) operations.

Brief Summary Text (20):
Other features include a second microcontroller and a second memory containing a second set of instructions configured to cause the second microcontroller to manage

device operations; a third microcontroller and a third memory containing a third set of instructions configured to cause the third microcontroller to manage memory operations; a fourth controller and a fourth memory containing a fourth set of instructions configured to cause the fourth controller to manage data operations; the plurality of microcontrollers are connected together to communicate with each other; a cross-bar switch to connect the plurality of microcontrollers; wherein the memory is a read only memory (ROM); wherein the memory is an erasable programmable ROM (EPROM); wherein the memory is a Flash memory; wherein the microcontrollers are digital signal processors (DSPs); wherein the apparatus is contained in a semiconductor chip; a device including a fifth microcontroller and a fifth memory containing a fifth set of instructions configured to cause the fifth microcontroller to control the device operations, the fifth microcontroller in communication with at least the second microcontroller; the fifth set of instructions in the fifth memory further configured to cause the fifth microcontroller to test the device and if the device is operational the fifth set of instructions is configured to cause the fifth microcontroller to signal at least the second microcontroller to indicate availability of the device; the fifth set of instructions in the fifth memory further configured to cause the fifth microcontroller to signal at least the second microcontroller to indicate availability of the device includes sending device identification and required resource data; the second set of instructions in the second memory further configured to cause the second microcontroller to receive the signal indicating availability of the device and allocating available resources to the device.

Brief Summary Text (21):
In another aspect of the invention a device for use in a computer system that eliminates a need for an operating system comprises device circuitry, a first microcontroller and a memory containing a set of instructions configured to cause the microcontroller to control device circuitry, the instructions further configured to facilitate the microcontroller to communicate with a second microcontroller that manages a central processing unit (CPU) operation.

Detailed Description Text (2):
The invention relates to hardware/firmware layer in a computer system that eliminates a need for an operating system. In one aspect of the invention, the hardware/firmware layer comprises a Device Manager, an Information Manager, a Memory Manager, and a Process Manager, which may be contained in one or more semiconductor chips. Each of the Managers comprises a microcontroller associated with firmware embedded in ROM or Flash memory that contains instruction sets that cause the microcontroller to provide a designated task of device management, information management, memory management and process management. The semiconductor chip or chip sets, when incorporated into the computer motherboard allows the computer hardware to operate without an operating system. In another aspect of the invention, devices such as disk drives, modem, printer, video monitor, connected to the computer system are "smart devices," wherein each device has a device microcontroller and embedded device drivers in a ROM or Flash memory. The is a semiconductor chip used in a computer system without an operating system does not need to search for available devices, provide diagnostic tests or obtain device drivers to communicate with the devices. Instead the device microcontroller uses the embedded device driver to perform configuration and self diagnostic test autonomously as well as operations of the device. If the device is operational, the device microcontroller sends an identification signal that includes resource requirements to the semiconductor chip used in a computer system without an operating system chip via trace links on the motherboard to indicate availability of the device.

Detailed Description Text (5):
With reference to FIG. 6 and FIG. 9, the Device Manager 610 identifies all devices 650, 660, 670 connected to the computer system and establishes their means of connection, control unit functionality, and device capacity and capability as will

be apparent with respect to FIG. 14. The device management routine 614 provides the instructions that allow the Device Manager 610 to act as an I/O scheduler by allocating devices 650, 660, 670 to tasks, initiating operations by the device and reclaiming the device on task completion. The user's interaction with the computer's resources is via requests submitted to Device Manager 610 from input devices 650 such as a mouse or a keyboard supported by a graphic user interface (GUI) provided via Device Manager 610 to the user's screen.

Detailed Description Text (6):
With reference to FIG. 6 and FIG. 10, the Process Manager 620 acts on process requests submitted via Device Manager 610. Using the instructions provided by the process management routine 624, the Process Manager 620 directly allocates CPU 202 resource to each request based on its needs including the allocation of registers within the CPU 202 and main memory 206. The Process Manager 620 will also request the Device Manager 610 to process relevant data transfer to main memory 206 from the permanent storage device 670. Process Manager 620 also provides a continuous tracking of the CPU 202 capacity and the status of processes and will reclaim the CPU 202 for new activities as each process terminates or exceeds allocated resources.

Detailed Description Text (7):
With reference to FIG. 6 and FIG. 11, The memory management routine 634 provides the instructions for the Memory Manager 630 to allocate memory 206 resources to the Process Manager 620 based on the immediate memory needs of the jobs being controlled by Process Manager 620. The Memory Manager 630 also responds to direct requests via Device Manager 610 for memory 206 resource. The Memory Manager 630 constantly tracks the user and associated amount of each element of memory being utilized and on termination of jobs, reclaims and makes memory resources re-available. Memory Manager 630 logically partitions the memory resource into two areas. One is reserved for resident applications in use (e.g. Microsoft.TM. Word.TM.) and the other for process requirements (e.g. a datafile being worked on by Excel.TM. or a program being developed in C++). Applications in use are loaded sequentially into the application memory space. Files and data only occupy the process memory space when they are being operated on and the Memory Manager 630 prioritizes, schedules and maintains an external queue for memory resource.

Detailed Description Text (8):
With reference to FIG. 6 and FIG. 12, the information management routine 644 provides the instructions for the Information Manager 640 to provide the means for the user to make decisions about planned and current use of the computer system's facilities by providing a complete information resource record and management capability for the entire system. Using input from the other Managers and directly from connected permanent storage devices 670, it maintains a file management record file in the main memory 206. This contains the location, use and size of all files, the status of all files (i.e. open/close, file type, security levels) and the overall capacities and usage levels of memory 206, input devices 650, output devices 660 and permanent storage devices 670.

Detailed Description Text (12):
A smart device 1400, illustrated in FIG. 14, contains a device microcontroller 1402 and embedded device driver 1404 within the device 1400 that allows it to configure and perform self diagnostic test as well as operations of the device, hence is smart in the sense that it operates independently without external control source. Smart devices include devices such as disk drives, modem, printer, video monitor and so forth. Smart devices are interfaced with the pertinent Manager in the chip used in the computer system without an operating system to make available the services of the devices. According to one aspect of the invention, at power-up initialization, individual device microcontrollers 1402 within the devices 1400 are activated, each device microcontroller 1402 having a startup address pointing to the embedded ROM contained in the device 1400. The ROM can contain a device driver

1404 that is particular to the device that the device microcontroller 1402 uses to configure and perform self diagnostic test. In one aspect of the invention, each device microcontroller 1402 in the one or more devices 1400 perform the configuration and diagnostics autonomously and simultaneously or substantially simultaneously. If, after the previous procedure, the device 1400 is deemed to be operational, the device microcontroller 1412 sends an identification signal 1500, which is stored in registers, such as the one illustrated in FIG. 15, to the chip 600 used in a computer system without an operating system via the trace link 1328 to indicate availability of the device 1400. The identification signal 1500 may be a string of data bits transmitted serially or in parallel that comprises of device identification bits 1502 and required resource data bits 1504. The signal may conform to the serial identifier and resource data format of the Plug and Play ISA specification. It should be noted that the signal configuration may be in any format standard established with or by device manufacturers. The Device Manager 610, upon receiving the identification signal, recognizes the presence of the device. Because the Device Manager 610 has access to available system resources, the Device Manager 610 identifies the resources required by the device and allocates the appropriate resources to the device. Resource allocation is usually performed in a manner that does not conflict with other devices that require system resources. The resource allocated device is assigned an address that acts as a pointer to the device. It should be noted that because the number of I/O and storage devices and the ability to communicate with them is predetermined as a function of the total peripheral device connectors, expansion slots and associated trace links to the chip 600, in one embodiment, device recognition and logical connectivity can be based on a finite state machine.

Detailed Description Text (13):
As can be seen by reference to FIG. 13, the chip 600 used in a computer system without an operating system is shown as being directly connected to the CPU (CPU slot 1303) and the main memory (memory slots 1302). Thus the CPU 202 and memory 206 (see FIG. 6) can focus entirely on process requirements and multi-tasking management that are performed by the Process Manager 620 and the Memory Manager 630 within the chip 600. As a result, when insufficient memory or CPU capacity is available, then no more processes are added to the system and the request is rejected at the input device level rather than causing a potential crash at the system level. As a result of separating processing and memory management from other routines, the processing speed is a straightforward relationship between the stated CPU 202 capacity, the memory 206 capacity and the transfer rate between and amongst the chip 600, the CPU 202 and the memory 206. This serves to provide robustness in the computer architecture as well as faster interaction between the CPU 202 and memory 206 as the CPU 202 need not be diverted to process other miscellaneous routines.

Detailed Description Text (20):
At stage 180, Information Manager 640 loads a file containing the recognized system state on previous shut-down. This contains pointers towards the record fields for all available programs and data files on each permanent storage device 670. At stage 182, the Information Manager 640 passes control to the Process Manager 620 along with an instruction to load the graphical user interface (GUI) from a predetermined fixed drive location. Process Manager 620 displays the GUI via the Device Manager 610 to the screen of the video monitor. The Information Manager 640 icon is displayed on the screen along with icons representing data or program files which the user had screen displayed prior to shut down. The chip 600 used in a computer system without an operating system is now ready to recognize and act on user requests or other inputs. An interrogation of the Information Manager 640 would reveal the GUI, record field file and device record file resident in memory, which would indicate the devices connected and their current status.

Detailed Description Text (21):
FIG. 18 and FIG. 19 illustrate an exemplary procedure for running program/software

applications. At stage 192, the user inputs this request typically via input device 650 such as mouse and/or keyboard and their interface with Device Manager 610. At stage, 194, the request is sent to Process Manager 620, which in combination with Memory Manager 630, identifies the appropriate files from the register in the memory, which identifies if sufficient memory 206 and CPU 202 resource is available, allocates it and requests the files from Device Manager 610, at stage 196. At stage 198, Device Manager 610 directs the relevant files from the device to memory. At stage 202, program file identification occurs between the Process Manager 620 and the Memory Manager 630. At stage 204, the Memory Manager 630 keeps track of the memory management, allocation of memory used by the programs. The program runs until it is complete or exceeds the resource allocated. Storage requirements during the program run are coordinated between the Device Manager 610 and the Memory Manager 630, at stage 206. In-process file request, queuing and delivery is coordinated between the Process Manager 620 and the Memory Manager 630, stage 208. At stage 212, if further user input is required, Process Manager 620 requests this of the user via the Device Manager 610 interface. The Information Manager 640 provides the user with a picture of the remaining system resources if required, at stage 214.

Detailed Description Text (22):
FIG. 18 and FIG. 19 also illustrate an exemplary procedure of concurrent processing. Inter-process communication and instruction, (e.g. Micrsoft Excel, a spread sheet program that requests the Process Manager 620 to run Microsoft Powerpoint, a presentation program) is handled in a similar manner. At stage 208, an inter-process request queue is created on a permanent storage device 670 and these are then scheduled as input requests in the above manner. As the management of program takes place outside of the CPU 202 and memory 206, the ability to continue to load and run more and more program is a straightforward function of comparing their combined memory 202 and CPU 202 requirement with the programs requirement. The Process Manager 620 and Memory Managers 630 conduct this task. The Memory Manager 630 also provides the Device Manager 610 with the information required to queue data on permanent storage prior to allocation of memory at the time of processing, stage 206. In an operating systemless approach the criteria for subsequently reclaiming the processor resource is pre-defined prior to its allocation to a particular processing requirement. This requirement is either completed and the next process is loaded or it exceeds its pre-allocated resource and is interrupted. Information Manager 640 provides the user with a continuous status indication and allows advance recognition of potential system overload. In addition, an inadvertent attempt to overload the system will simply be met with a message indicating that the process cannot be performed. As the CPU 202 and memory 206 are not engaged at this point, there is no danger of a system crash.

CLAIMS:

1. A computer system that eliminates a need for an operating system comprising: a central processing unit (CPU); a main memory; at least one device; a further unit that includes a first microcontroller a first memory containing a first set of instructions configured to cause the microcontroller to manage CPU operations, a second microcontroller in communication with the first microcontroller; and a second memory containing a second set of instructions configured to cause the second microcontroller to manage the at least one device; a third microcontroller in communication with the first microcontroller; a third memory containing a third set of instructions configured to cause the third microcontroller to manage memory operations; a fourth microcontroller in communication with the first microcontroller; and a fourth memory containing a fourth set of instructions configured to cause the fourth controller to manage data operations; and a plurality of trace links connecting the further unit to the CPU, the main memory, and the device interface to facilitate communication between the further unit, the CPU, the main memory, and the device, the device comprising: a fifth microcontroller in communication with at least the second microcontroller; and a

fifth memory containing a fifth set of instructions in the fifth memory configured to cause the fifth microcontroller to test the device and if the device is operational the fifth set of instructions is configured to cause the fifth microcontroller to signal at least the second microcontroller to indicate availability of the device.

4. The computer system as in claim 1, wherein the fifth set of instructions in the fifth memory is further configured to cause the fifth microcontroller to signal at least the second microcontroller to indicate availability of the device includes sending device identification and required resource data.

5. The computer system as in claim 4, wherein the second set of instructions in the second memory is further configured to cause the second microcontroller to receive the signal indicating availability of the device and allocating available resources to the device.

6. The computer system of claim 1, wherein the each microcontroller is associated with firmware and wherein the first, second, third, fourth, and fifth instruction sets are contained in associated firmware.

Hit List

Clear | Generate Collection | Print | Fwd Refs | Bkwd Refs | Generate OACS

Search Results - Record(s) 1 through 2 of 2 returned.

☐ 1. Document ID: US 6591358 B2

L22: Entry 1 of 2 File: USPT Jul 8, 2003

US-PAT-NO: 6591358
DOCUMENT-IDENTIFIER: US 6591358 B2

TITLE: Computer system with operating system functions distributed among plural microcontrollers for managing device resources and CPU

DATE-ISSUED: July 8, 2003

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Jaffrey; Syed Kamal H.	Watertown	MA	02472	

APPL-NO: 09/ 770810 [PALM]
DATE FILED: January 26, 2001

INT-CL: [07] G06 F 13/12

US-CL-ISSUED: 712/32; 700/3, 700/19, 709/327, 710/8, 710/15, 710/104, 713/1, 713/2
US-CL-CURRENT: 712/32; 700/19, 700/3, 710/104, 710/15, 710/8, 713/1, 713/2, 719/327

FIELD-OF-SEARCH: 700/3, 700/19, 710/8, 710/15, 710/104, 712/32, 713/1, 713/2, 709/327

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
4831525	May 1989	Saito et al.	364/300
4961131	October 1990	Ashida	700/23
5412798	May 1995	Garney	709/321
5428787	June 1995	Pineau	713/1
5490272	February 1996	Mathis et al.	709/108
5526523	June 1996	Straub et al.	713/100
5559965	September 1996	Oztaskin et al.	710/104
5574886	November 1996	Koike et al.	395/500
5634074	May 1997	Devon et al.	710/8

h e b b g e e e f e e ef b e

5715456	February 1998	Bennett et al.	713/2
5742825	April 1998	Mathur et al.	709/329
5748980	May 1998	Lipe et al.	710/8
5802265	September 1998	Bressoud et al.	714/11
5819112	October 1998	Kusters	710/36
5835964	November 1998	Draves et al.	711/207
5878276	March 1999	Aebli et al.	395/839
5935224	August 1999	Svancarek et al.	710/63
5937200	August 1999	Frid et al.	710/264
5960087	September 1999	Tribble et al.	713/167
5999989	December 1999	Patel	710/1
6049854	April 2000	Bedarida	711/153
6061695	May 2000	Slivka et al.	715/513
6105074	August 2000	Yokote	709/328
6108715	August 2000	Leach et al.	709/330
6145021	November 2000	Dawson et al.	710/8
6154836	November 2000	Dawson, III	713/1
6154838	November 2000	Le et al.	713/2
6189049	February 2001	Klein	710/1
6189050	February 2001	Sakarda	710/18

OTHER PUBLICATIONS

Compaq Computer Corporation, Phoenix Technologies, Ltd. Intel Corporation, Plug and Play BIOS Specification, version 1.0A, May 5, 1994.
Plug and Play ISA Specification, version 1.0a, May 5, 1994.

ART-UNIT: 2183

PRIMARY-EXAMINER: Kim; Kenneth S.

ATTY-AGENT-FIRM: Mintz, Levin, Cohn, Ferris, Glovsky and Popeo, P.C.

ABSTRACT:

A hardware/firmware layer comprising a Device Manager, an Information Manager, a Memory Manager, and a Process Manager contained in one or more semiconductor chips is disclosed. The hardware/firmware layer eliminates the need for an operating system. Each of the Managers comprises a microcontroller associated with a firmware embedded in ROM or Flash memory that contains instruction sets that cause the microcontroller to provide a designated task of device management, information management, memory management and process management. In another aspect of the invention, devices connected to the computer system are "smart devices," each device having a device microcontroller and embedded device drivers in a ROM or Flash memory. The hardware/firmware of the present invention does not need to search for available devices, provide diagnostic tests or obtain device drivers to communicate with the devices. Instead, the device microcontroller uses the embedded device driver to perform configuration and self diagnostic test as well as device operations. If the device is operational, the device microcontroller sends an identification signal to the hardware/firmware layer of the present to indicate availability of the device.

7 Claims, 25 Drawing figures

☐ 2. Document ID: US 5838968 A

L22: Entry 2 of 2 File: USPT Nov 17, 1998

US-PAT-NO: 5838968
DOCUMENT-IDENTIFIER: US 5838968 A

TITLE: System and method for dynamic resource management across tasks in real-time operating systems

DATE-ISSUED: November 17, 1998

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Culbert; Daniel	Los Altos	CA		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Chromatic Research, Inc.	Sunnyvale	CA			02

APPL-NO: 08/ 609337 [PALM]
DATE FILED: March 1, 1996

INT-CL: [06] G06 F 9/00

US-CL-ISSUED: 395/674; 395/673, 395/675
US-CL-CURRENT: 718/104; 718/103, 718/105

FIELD-OF-SEARCH: 395/674, 395/675, 395/650, 395/67, 395/800, 364/244, 364/244.3, 364/281.6, 364/281.3

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
4727487	February 1988	Masui et al.	395/67
4890227	December 1989	Watanabe et al.	395/800
5233533	August 1993	Edstrom et al.	364/468
5325525	June 1994	Shan et al.	395/650

OTHER PUBLICATIONS

Sudarshan K. Dhall and C.L. Liu, "On a Real-Time Scheduling Problem", Operations Research, vol. 26, No. 1, Jan.-Feb. 1978, pp. 127-140.
C.L. Lou and James W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", Journal of the Association for Computing Machinery, vol. 20, No. 1, Jan. 1978, pp. 46-61.

h e b b g e e e f e e ef b e

J.F. Bortolotti, P. Vernard, and E. Bouchet, "RTKM: A Real-Time Microkernel", Dr. Dobb's Journal, May 1994, pp. 70, 72, 74, 76, 105-106.
David Shear, "Three DSP RTOSs are Ready to Merge with Windows", EDN-Technology Update, Jun. 1994, pp. 29-30, 32 and 34.

ART-UNIT: 275

PRIMARY-EXAMINER: Banankhah; Majid A.

ATTY-AGENT-FIRM: Skjerven, Morrill, MacPherson, Franklin & Friel, L.L.P. Terrile; Stephen A.

ABSTRACT:

A system and method for dynamic resource management across tasks in real-time operating systems is disclosed. The system and method manage an arbitrary set of system resources and globally optimize resource allocation across system tasks in a dynamic fashion, according to a system specified performance model. The present invention provides a mechanism for system programmers to program tasks such that system performance will be globally optimized and dynamically managed over a system programmer-controllable set of system resources. The invention supports a mechanism for defining and managing arbitrary resources through a task resource utilization vector. Each task resource utilization vector contains an arbitrary number of task resource utilization records that contain quantities of system resources that each task qualitatively prefers to utilize while executing on the processor. Each of the task utilization records contains a run level that reflects the associated task's ability to perform its work when allocated the resources according to the particular task resource utilization record. This run level is used to dynamically vary the quantity of system resources that the task has allocated, based on the availability of system resources and the priorities of the tasks.

23 Claims, 7 Drawing figures

Term	Documents
(20 AND 21).USPT.	2
(L21 AND L20).USPT.	2

h e b b g e e e f e e ef b e

L20: Entry 2 of 5 File: USPT Jun 5, 2001

DOCUMENT-IDENTIFIER: US 6243735 B1
TITLE: Microcontroller, data processing system and task switching control method

Abstract Text (1):
A processor, a task management table, and a scheduler are built in a microcontroller. The processor sequentially runs a plurality of tasks for controlling hardware engines (cores) respectively allocated thereto. The task management table stores task management information which includes state information (ST INFO) representative of the execution state of each task, priority information (PRI INFO) representative of the execution priority of each task, and core identification information (CID INFO) representative of the allocation of the tasks to the cores. The scheduler allows the processor to switch between tasks on the basis of the task management information when a given instruction is decoded or when the execution of any one of the cores is terminated.

Brief Summary Text (2):
This invention relates to a multitasking microcontroller and to a data processing system employing such a multitasking microcontroller operable to control a plurality of hardware engines and it further relates to a method of controlling task switching.

Brief Summary Text (3):
Multitasking microcontrollers have been known in the art. In a typical multitasking microcontroller, a built-in processor sequentially executes a plurality of tasks and a task timer therefore makes periodic issues of timer interruptions which request task switching to be made. Every time the processor accepts such a timer interruption, an interruption handling routine in the operating system (OS) is activated. The interruption handling routine performs the scheduling of tasks and the saving and restoring of resources.

Brief Summary Text (4):
Conventional multitasking microcontrollers have some drawbacks. For example, the scheduling of tasks is carried out using an interruption handling routine in a conventional multitasking microcontroller. This produces the problem that there is created much overhead at task switching time, therefore resulting in a drop in microcontroller performance.

Brief Summary Text (10):
The objects of the present invention are achieved as follows. The microcontroller of the present invention controls the task switching not by means of an interruption handling routine, but by means of a hardware scheduler. In accordance with the present invention, a plurality of tasks are allocated to respective hardware engines. In such an environment, a task switching operation is controlled by the hardware scheduler on the basis of information representative of the allocation of the tasks to the hardware engines. Some of the hardware engines execute time critical processes and the other hardware engines do not. In accordance with the present invention, a relationship among the hardware engines is reflected in the execution priority of the tasks, which makes it possible to select a task to be run next in a short time without redetermining which of the hardware engines executes a time critical process at the time of task switching. In other

words, there is created less overhead when task switching occurs. High-speed task switching is realized.

Brief Summary Text (11):
Undesirable dead time occurs in a time sharing method which carries out switching between tasks in response to an interruption periodically issued by a task timer, in view of which the present invention was made. Accordingly, the microcontroller of the present invention adopts an event-driven method capable of performing task switching in fast response to the occurrence of an event (i.e., an event of hardware engine execution termination). Each task can be in one of at least three states: a first state (the state of READY) representative of an execution wait status, a second state (the state of ACTIVE) representative of a running status, and a third state (the state of SLEEP) representative of an allocated hardware engine execution termination status. A task is in the state of ACTIVE when it uses the microcontroller and in the state of ACTIVE a hardware engine allocated to the task is controlled. A task is in the state of READY when it is not selected therefore waiting to be selected although it is ready to use the microcontroller. A task is in the state of SLEEP when it waits for a hardware engine allocated thereto to be execution-terminated (in other words, it is not ready to use the microcontroller). A task that has finished activating its allocated hardware engine makes a state transition from ACTIVE to SLEEP in response to a given instruction (the task.sub.13 sleep instruction). When the execution of a certain hardware engine is terminated, a task allocated to that hardware engine makes a state transition from SLEEP to READY and a task under execution makes a state transition from ACTIVE to READY. Thereafter, a task having the highest execution priority in all tasks assuming the state of READY is selected as a task to be run next. The task thus selected makes a state transition from READY to ACTIVE.

Brief Summary Text (12):
If a plurality of register files are prepared in the microcontroller so that a plurality of hardware engines can use the register files as mutually independent working areas, there is created much less overhead at the time of task switching because what is required to do at the task switching time is just saving processor resources such as program counter. A register file for storing a setting parameter common to a plurality of hardware engines can be prepared in the microcontroller.

Detailed Description Text (3):
FIG. 2 shows in detail the structure of the microcontroller 101. The microcontroller 101 has a task controller 201 for the realization of multitasking, five core register files 211-215 for use by the five cores 111-115 as mutually independent working areas, a single common register file 216 for storing a setting parameter common to at least two of the five cores 111-115, a single general-purpose register file 217 for use by the task controller 201 as a working area, a multiplier 221, a shifter 222, an arithmetic and logic unit (ALU) 223, and a data memory 224. 241 is an A bus. 242 is a B bus. 243 is a C bus. 231 is a signal line for connecting together the buses 241-243 and the task controller 201. The task controller 201 provides the activation signal 123 and receives the termination signal 124. Each of the register files 211-216 is connected between the C bus 243 and a corresponding one of the signal lines 131-136. Additionally, each of the register files 211-216 has two outputs that are connected to the A bus 241 and to the B bus 242 respectively. The general-purpose register file 217 and the data memory 224 each have a single input that is connected to the C bus 243 and two outputs that are connected to the A bus 241 and to the B bus 242 respectively. The multiplier 221, the shifter 222, and the ALU 223 each have two inputs that are connected to the A bus 241 and to the B bus 242 respectively and a single output that is connected to the C bus 243. A variation to the above can be made in which the placement of the five core register files 211-215 and the common register file 216 is omitted and the signal lines 131-136 extend directly form the C bus 243.

Detailed Description Text (5):

h e b b g e e e f c e e e ge

The above is discussed in detail with reference to FIG. 2. The task controller 201 first sets an operating parameter to the MD core register file 211 through the signal line 231, the ALU 223, and the C bus 243 and provides the activation signal 123 to make the MD core 111 active. The MD core 111 reads in the operating parameter from the MD core register file 211 through the signal line 131 and inputs the image data 121. Upon termination of the execution of the MD core 111, candidate motion vectors found in the MD core 111 are written into the MD core register file 211 through the signal line 131, and the MD core 111 provides the termination signal 124. In response to the termination signal 124, the task controller 201 reads out the candidate motion vectors from the MD core register file 211. Based on the candidate motion vectors, the task controller 201 computes an operating parameter for the MC core 112 by the use of the multiplier 221, the shifter 222, the ALU 223, and the general-purpose register file 217. The operating parameter is set to the MC core register file 212 and the MC core 112 is made active by the activation signal 123. The MC core 112 reads in the operating parameter from the MC core register file 212 through the signal line 132. Thereafter, the MC core 112 finds image differential data. Upon termination of the execution of the MC core 112, a sum of the image differential data is written into the MC core register file 212 by way of the signal line 132, the image differential data are written into the buffer memory 116, and the MC core 112 provides the termination signal 124. In response to the termination signal 124, the task controller 201 reads out the image differential data sum from the MC core register file 212. Based on the image differential data sum, the task controller 201 selects an optimum motion vector from among the aforesaid candidate motion vectors through the use of the multiplier 221, the shifter 222, the ALU 223, and the general-purpose register file 217. An address indicative of the location of differential data corresponding to the optimum motion vector is set in the DCT core register file 213 and the DCT core 113 is made active by the activation signal 123. Based on the address set in the DCT core register file 213, the DCT core 113 reads out the differential data from the buffer memory 116 for DCT. Upon termination of the execution of the DCT core 113, a result of the DCT operation is written into the buffer memory 117 and the DCT core 113 provides the termination signal 124. The Q core 114 performs quantization and a result of the quantization operation is written into the buffer memory 118. The VLC core 115 performs VLC and a result of the VLC operation is provided as the encoded data 122. Some of the five cores 111-115 exchange the signals 123 and 124 with the microcontroller 101 a plurality of times per macroblock processing. The common register file 216 is used in cases such as when a common parameter for switching between MPEG1 and MPEG2 is pre-supplied to the five cores 111 and when a common parameter for designating a motion estimation mode is pre-supplied to the cores 111 and 112.

Detailed Description Text (6):
Referring to FIG. 3, the structure of the task controller 201 is now described in detail. The task controller 201 has a processor 300, a task management table 310, and a scheduler 330. The processor 300 is a RISC (reduced instruction set computer) processor capable of sequential execution of eight tasks at most. The processor 300 has a program counter (PC) 301 for generating instruction addresses, an instruction memory 302 for storing a program of a series of instructions, and an instruction decoder 303 for decoding instructions. The instruction decoder 303 sends the activation signal 123 to each core. The instruction decoder 303 is connected to resources for the execution of instructions, such as the multiplier 221, the shifter 222 and the ALU 223, through the signal line 231. The task management table 310 is a circuit block for storing task management information. The task management table 310 has eight task management register files 320-327 that are associated with eight tasks from TASK0 to TASK7, respectively. The task management information includes state information (ST INFO) representative of the execution status of each task, priority information (PRI INFO) representative of the execution priority of each task, and core identification information (CID INFO) representative of the allocation of the tasks to the five cores 111-115. Additionally, the task management table 310 has PC regions for the tasks for saving the processor's 300

resources (i.e., the contents of the PC 301). Such a region is also used to save a flag concerning a result of the arithmetic operation of the ALU 223 (see FIG. 2). The scheduler 330 is a circuit block operable to allows the processor 300 to switch between tasks on the basis of the task management information stored in the task management table 310. The scheduler 330 has a state controller 331, a terminated core determination unit (TCDU) 332, a priority encoder 333, and a selector 334. In response to the termination signal 124 sent from any one of the five cores 111-115 (i.e., a core the execution of which is terminated), the TCDU 322 identifies a task allocated to that execution-terminated core. Such identification is carried out with reference to the task management table 310 and a task number 362 representative of a result of the identification operation is communicated to the state controller 331. The priority encoder 333 is a circuit block for selecting a task to be run next. Referring to the task management table 310, the priority encoder 333 performs such a selection operation and a task number 361 representative of a result of the selection operation is communicated to the state controller 331 as well as to the selector 334. The state controller 331 is a circuit block for updating the ST INFO stored in the task management table 310. The selector 334 controls the restoration of resources to the processor 300.

Detailed Description Text (7):
FIG. 4 illustrates an association between the cores and the tasks in the MPEG image encoder of FIG. 1. Here, the microcontroller 101 executes six tasks 400-405. The task 400 controls the five tasks 401-405 lower in hierarchy than the task 400 and is a main task (TASK0) for managing the entire encoding processing. The main task 400 is assigned no core. The task 401 is a motion detection task (TASK1) for controlling the allocated MD core 111. The task 402 is a motion compensation task (TASK2) for controlling the allocated MC core 112. The task 403 is a DCT task (TASK3) for controlling the allocated DCT core 113. The task 404 is a Q task (TASK4) for controlling the allocated Q core 114. Finally, the task 405 is a VLC task (TASK5) for controlling the allocated VLC core 115.

Detailed Description Text (8):
Suppose here that the task management table 310 of FIG. 3 stores task management information concerning at least six tasks (the six tasks 400-405). Referring to FIG. 3, the PRI INFO is set according to a priority setting signal 342. The CID INFO is set according to a core setting signal 343. The priority setting signal 342 is sent to the task management table 310 from the instruction decoder 303 if the instruction decoder 303 decodes a priority setting instruction. On the other hand, the core setting signal 343 is sent to the task management table 310 from the instruction decoder 303 if the instruction decoder 303 decodes a core setting instruction.

Detailed Description Text (9):
FIG. 5 is a conceptual diagram showing the state transition of each task. Each task can be in one of four states, namely, the state of STOP representative of a suspended status, the state of READY representative of an execution wait status, the state of ACTIVE representative of a running status, and the state of SLEEP representative of an allocated hardware engine execution termination wait status. SLEEP cannot exist for TASK0. If the task controller 201 is reset, it will assume the state of STOP for all tasks. A task in the state of STOP is changed to READY by a task_ready instruction (in other words, a transition 501 is made). If a task in the state of READY is selected by the scheduler 330 when an event requesting for task switching to be made occurs, the task is changed to ACTIVE (in other words, a transition 511 is made), at which time a task which has been placed in the state of ACTIVE up to the moment is changed to READY by the scheduler 330 (in other words, a transition 522 is made). A task in the state of ACTIVE is executed by the processor 300. A task in the state of ACTIVE can be changed to SLEEP by a task_sleep instruction (in other words, a transition 521 is made), alternatively it can be changed to STOP by a task_stop instruction (in other words, a transition 523 is made). A task in the state of SLEEP is changed to READY (in other words, a

transition 531 is made) if the execution of a core allocated to that task is terminated.

Detailed Description Text (10):
Details of the operation of the task controller 201 of FIG. 3 are described here. Task switching occurs if the instruction decoder 303 decodes the task_ready instruction, the task_sleep instruction, or the task_stop instruction. For example, when a certain task is run to finish setting an operating parameter for a core allocated to the task and activating the core, the state of the task is changed from ACTIVE to SLEEP by the task_sleep instruction. Additionally, task switching occurs upon termination of the execution of any one of the five cores 111-115. The operating sequence of the task controller 201 at task switching time includes (1) activating the scheduler (SCHEDULER ACTIVATION), (2) saving the resources of a task under execution (RESOURCE SAVING), (3) selecting a task to be run next (TASK SELECTION), and (4) restoring the saved resources (RESOURCE RESTORATION).

Detailed Description Text (11):
Firstly, a task switching sequence on the basis of instructions is explained.

Detailed Description Text (12):
(A-1) SCHEDULER ACTIVATION

Detailed Description Text (13):
If the task_ready instruction, the task_sleep instruction, or the task_stop instruction is decoded, the instruction decoder 303 provides a state change signal 341. The state change signal 341 is sent to the state controller 331. As a result, the scheduler 330 is made active.

Detailed Description Text (15):
The state change signal 341 is also sent to the task management table 310 and the ST INFO is updated. At the same time, the value of the PC 301 of a task which has been executed up to the moment is saved in the task management table 310 through a signal line 344.

Detailed Description Text (17):
The priority encoder 333 receives from the task management table 310 the ST INFO and the PRI INFO through a signal line 351 and through a signal line 352 respectively, to select a task having the highest priority of execution in all tasks that are in the state of READY as a task to be run next. The task number 361 indicative of a result of the task selection operation is communicated to the state controller 331 and to the selector 334.

Detailed Description Text (19):
The state controller 331 sends to the task management table 310 a state change signal 364 according to the task number 361. As a result, the ST INFO of the task selected by the priority encoder 333 is updated from READY to ACTIVE. The selector 334 reads out the PC of the task designated by the task number 361 from the task management table 310 through a signal line 353, for forwarding onto a signal line 363. As a result, the value of the PC of the task to be run next is set in the processor 300 and the execution of the task starts.

Detailed Description Text (21):
(B-1) SCHEDULER ACTIVATION

Detailed Description Text (22):
If the execution of any one of the cores is terminated, the termination signal 124 is sent to the TCDU 332. The TCDU 332 determines which of the cores is execution-terminated on the basis of the termination signal 124. Further, the TCDU 332 reads out the CID INFO stored in the task management table 310 through a signal line 354 and determines which of the tasks is allocated to the execution-terminated core.

The task number 362 indicative of a result of the determination operation by the TCDU 332 is communicated to the state controller 331 if the task in question is confirmed to be in the state of SLEEP from the ST INFO, in consequence of which the scheduler 330 is activated. The state controller 331 sends to the task management table 310 the state change signal 364 according to the task number 362, as a result of which the ST INFO of the execution-terminated task is updated from SLEEP to READY. The scheduler 330 will not be activated if there exists no task allocated to the execution-terminated core.

Detailed Description Text (24):
The state controller 331 sends the state change signal 364 to the task management table 310 so that the ST INFO of a task that has been under execution up to the moment is updated from ACTIVE to READY. At the same time, the value of the PC 301 of the task is saved in the task management table 310.

Detailed Description Text (26):
The priority encoder 333 receives the ST INFO and the PRI INFO from the task management table 310 thereby selecting a task having the highest priority of execution in all tasks in the state of READY as a task to be run next. The task number 361 indicative of a result of the task selection operation is communicated to the state controller 331 as well as to the selector 334.

Detailed Description Text (28):
The state controller 331 sends to the task management table 310 the state change signal 364 according to the task number 361. The ST INFO of the task selected by the priority encoder 333 is updated from READY to ACTIVE. The selector 334 reads out the PC of the task designated by the task number 361 from the task management table 310, for forwarding to the processor 300. As a result, the value of the PC of the task to be run next is set in the processor 300 and the execution of the task in question starts.

Detailed Description Text (30):
FIG. 7 shows the state transition of each of three tasks in a part period specified by broken line of FIG. 6. TASK0 is a main task for managing an entire encoding process, TASK1 is a task allocated to the MD core 111, and TASK2 is a task allocated to the MC core 112 (see FIG. 4). In these three tasks, TASK1 has the highest execution priority. TASK2 has the second highest execution priority. TASK0 has the lowest execution priority. Suppose that at time t0, TASK1 is in the state of ACTIVE and TASK0 and TASK2 are in the state of READY.

Detailed Description Text (31):
FIG. 7 shows that task switching occurs at each time t1-t7. In FIG. 7, At represents the overhead of one task switching operation. The description will be made in order. TASK1 makes the MD core 111 active prior to time t1. At time t1, the state of TASK1 is changed from ACTIVE to SLEEP by the task_sleep instruction. At this point in time, although TASK0 and TASK2 are in the state of READY, it is TASK2 that is allowed to make a transition from READY to ACTIVE, since TASK2 has priority of execution over that of TASK0. TASK2 makes the MC core 112 active. Then, at time t2, the state of TASK2 is changed from ACTIVE to SLEEP by the task_sleep instruction. At this point in time, only TASK0 assumes the state of READY. TASK0 therefore makes a state transition from READY to ACTIVE. At time t3, the state of TASK2 is changed from SLEEP to READY by the execution termination of the MC core 112 and the state of TASK0 (which has been in the state of ACTIVE up to the moment) moves to READY. At this point in time, although TASK0 and TASK2 are in the state of READY, it is TASK2 that is allowed to make a state transition from READY to ACTIVE, since TASK2 has priority of execution over that of TASK0. TASK2 again activates the MC core 112. At time t4, the state of TASK2 is changed from ACTIVE to SLEEP by the task_sleep instruction, at which point in time only TASK0 is in the state of READY. TASK0 therefore makes a state transition from READY to ACTIVE. At time t5, the state of TASK1 is changed from SLEEP to READY by the execution termination of the

MD core 111 and the state of TASK0 (which has been in the state of ACTIVE up to the moment) moves to READY. At this point in time, although TASK0 and TASK1 are in the state of READY, it is TASK1 that is allowed to make a state transition from READY to ACTIVE, since TASK1 has priority of execution over that of TASK0. At time t6, the state of TASK2 is changed from SLEEP to READY by the execution termination of the MC core 112 and the state of TASK1 (which has been in the state of ACTIVE up to the moment) moves to READY. At this point in time, although TASK0, TASK1, and TASK2 are all in the state of READY, it is TASK1 that is allowed to return to ACTIVE from READY, since TASK1 has the highest execution priority in all the tasks. TASK1 again makes the MD core 111 active. At time t7, the state of TASK1 is changed from ACTIVE to SLEEP by the task_sleep instruction. At this point in time, although TASK0 and TASK2 are in the state of READY, it is TASK2 that is allowed to make a state transition from READY to ACTIVE, since TASK2 has priority of execution over that of TASK0.

CLAIMS:

1. A microcontroller comprising:

(a) a processor for sequentially executing a plurality of tasks in accordance with programmed instructions, said processor operating in conjunction with a plurality of hardware engines;

(b) a task management table for storing task management information including (1) state information representative of the execution status of each said task, (ii) priority information representative of the execution priority of each said task, and (iii) allocation information representative of the allocation of said plurality of tasks to said plurality of hardware engines; and

(c) a scheduler for allowing, on the basis of said task management information, said processor to switch between tasks, wherein each said plurality of hardware engines starts execution of a data process upon the activation by said processor and, if said data process is terminated, informs said scheduler of the termination of execution, and said scheduler allows said processor to switch between tasks if the termination of execution of any one of said hardware engines is detected.

3. The microcontroller according to claim 2, wherein when during execution of a task of said plurality of tasks said processor activates a hardware engine allocated to said task under execution before decoding a given instruction, said processor performs a function of updating said state information so that said task makes a state transition from said second state to said third state.

4. The microcontroller according to claim 2,

said scheduler including:

a determination unit for identifying, when the execution of any one of said plurality of hardware engines is terminated, a task allocated to said execution-terminated hardware engine on the basis of said task management information; and

a state controller which performs, upon being activated by said determination unit, a function of updating said state information so that said identified task makes a state transition from said third state to said first state.

6. The microcontroller according to claim 4, wherein said scheduler further includes a priority encoder for selecting, on the basis of said task management information, a task having the highest execution priority in all tasks that are in said first state as a task to be run next.

8. The microcontroller according to claim 1, wherein said task management table has

a region in which to save resources of said processor concerning a task that was run prior to the occurrence of the aforesaid task switching.

9. The microcontroller according to claim 1 further comprising a plurality of register files for use by said plurality of hardware engines as mutually independent working areas.

10. The microcontroller according to claim 1 further comprising a register file used to store a setting parameter common to at least two of said plurality of hardware engines.

11. A data processing system comprising:

a plurality of hardware engines for executing respective data processes; and

a microcontroller for controlling said plurality of hardware engines;

said microcontroller including:

a processor for sequentially executing a plurality of tasks in accordance with programmed instructions, said processor operating in conjunction with said plurality of hardware engines;

a task management table for storing task management information including (i) state information representative of the execution status of each said task, (ii) priority information representative of the execution priority of each said task, and (iii) allocation information representative of the allocation of said plurality of tasks to hardware engines; and

a scheduler for allowing, on the basis of said task management information, said processor to switch between tasks, wherein each said plurality of hardware engines starts execution of a data process upon the activation by said processor and, if said data process is terminated, informs said scheduler of the termination of execution, and said scheduler allows said processor to switch between tasks if the termination of execution of any one of said hardware engines is detected.

14. A task switching control method including allocating one or more tasks to corresponding hardware engines and controlling, based on information about such task/hardware engine allocation, task switching by the use of a scheduler, wherein:

each said hardware engines starts execution of a data process upon the activation by a processor and, if said data process is terminated, informs said scheduler of the termination of execution;

each said task can be in one of a first state representative of an execution wait status, a second state representative of a running status, and a third state representative of a wait status awaiting the termination of execution of a hardware engine allocated thereto; and

when the execution of any one of said hardware engines is terminated, said scheduler changes the state of a task allocated to said executon-terminated hardware engine from said third state to said first state so as to allow said processor to switch between tasks.

L44: Entry 6 of 11 File: USPT Nov 30, 1999

DOCUMENT-IDENTIFIER: US 5995997 A
TITLE: Apparatus and methods for optimally allocating currently available computer resources to future task instances versus continued execution of current task instances

Brief Summary Text (5):
Generally speaking, a single computer processor, while executing a given software program and under control of an associated operating system, executes a series of atomistic processes--which collectively implement the program. However, in any single sequential Von Neumann type processor, only one such process executes at any one time. A significant response time imbalance often exists between a rate at which the processor itself executes instructions for a given process and the substantially slower response time of a resource, e.g. a hard disk drive or other mass storage facility, utilized by that processor. Hence, when the process requires use of such a resource, the operating system issues a request for the resource and then ceases executing the process, i.e. the process is suspended and simply waits, until the resource has properly responded to the request. This wait time can become significant if the resource is busy, i.e. unavailable, at the time the request is issued. While the process itself could not advance its execution pending the availability of the resource, the computer processor itself is often not so constrained.

Brief Summary Text (24):
In accordance with the broad teachings of the present invention, presently available processing resources, here illustratively processing time, are allocated, within a computer system, to a task instance that will provide the largest "value" (or benefit), i.e. current fixed or incremental utility or future cost saving, to a current user of that system. Such an allocation can occur during idle-time intervals, in order to select a future task instance(s) most suited for precomputation thereduring, and/or alternatively during other processing intervals, such as during intervals of high or low activity, to determine whether enhanced current value will arise by prematurely suspending a currently executing task instance in favor of precomputing a future task instance.

Brief Summary Text (28):
In accordance with my specific inventive teachings, the desirability of continuing execution of a currently executing task instance vis-a-vis prematurely suspending the refinement of that instance in favor of precomputing a future task instance is assessed through use of discounted net expected value (NEV) exhibited by that instance. NEV is determined as a product, of the probability of a future task instance and its EVC flux, multiplied by a suitable time-discount factor. The currently executing task instance is terminated, i.e. suspended, in favor of precomputing a future task instance if, at the onset of a time slice, the latter instance exhibits a discounted NEV that exceeds the EVC flux then being provided by the former instance. Precomputation continues for the remainder of the time slice, with a re-evaluation of discounted NEV (including that of the suspended task instance) as against the EVC flux provided by the task instance then currently executing at the beginning of each successive slice, and so forth, in order to optimally allocate currently available processing resources to their best current use.

Detailed Description Text (47):
I have recognized that in certain situations, a future task instance(s), if precomputed, could provide greater expected value than the value provided by a task which is currently executing. In that regard, a portion of current task execution could be intentionally degraded or that task instance completely suspended, hence retarding or even halting execution of that task and freeing processing capacity, in favor of allocating that capacity to such a future task instance. Considering the net present value of results to be delivered by precomputing a future task instance necessitates time-discounting the net expected value of these results to the present. The discounted net expected value is then compared to losses in current value that would result from reducing or prematurely suspending the refinement of a current task instance. If the discounted value for any such future task instance(s) exceeds these losses, then a portion of present processing capacity is diverted from the presently executing task in favor of precomputing the future task instance(s).

Detailed Description Text (48):
An immediate loss of dedicating current resources, such as processing time, over a period of time is the product of that resource and average EVC flux over that period. The gains of allocating these resources to a future task instance is a function of the total amount of idle-time that will be presently available after a current task is degraded or prematurely suspended. If this idle time is sufficient, in duration, to fully process not only the present task instance but also all possible future task instances, then essentially nothing can be gained by transferring current resources to precomputing any future task instance. Hence, the expected value needs to be modified by a probability distribution, in terms of duration, taken over the available idle time.

Detailed Description Text (55):
Hence, in accordance with my inventive teachings, I conclude that current resources, such as processing time, should not be allocated to a future task if the EVC flux provided by the currently executing task instance exceeds the largest incremental utility measure of that future task instance, i.e., for example, the largest product of the probability of a future task instance and the EVC flux of that future task instance. Therefore, if for any time slice, the EVC flux for the currently executing task instance exceeds a largest incremental utility measure (i.e., the product of the EVC flux and the probability of the future task instance), then the affect on this product of discounting as well as the probability associated with idle time are both irrelevant and need not be considered, inasmuch as precomputation would be inferior to continuing the task instance that is currently executing during this slice. Alternatively, if precomputation is warranted, i.e. the discounted product exceeds the EVC flux for the currently executing task instance, then execution of this current task instance is suspended (which here includes its termination or merely retarding its execution, as appropriate) and the future task instance having the largest product of probability and EVC flux is precomputed to the fullest extent possible.

Detailed Description Text (57):
For future task instances having non-linearly varying EVC, the EVC flux of the presently executing task is compared to the discounted incremental utility measure, i.e., here discounted EVC flux-probability product, of the future tasks, with precomputation occurring only if the latter exceeds the former. This comparison could occur at either of two intervals: either at regular time slices, as described above, or, for each future task instance that is being precomputed, at the conclusion of that future task instance. In the latter situation, precomputation would continue only if the discounted EVC flux-probability product for the future task exceeded the EVC flux of the current task then suspended. As additional future task instances are encountered, their discounted EVC flux-probability products would be considered in an identical fashion. Likewise, as a future task instance is

precomputed, its discounted EVC flux-probability product is removed from further consideration.

Detailed Description Text (59):
To enhance understanding of the concept of allocating resources to precompute a future task instance at the cost of prematurely suspending a presently executing task instance, the reader should now consider FIG. 4.

Detailed Description Text (60):
This figure graphically depicts illustrative non-linearly varying .phi..times.p curve 410 for a currently executing task instance and illustrative linearly varying discounted .phi..times.p curve 450 associated with a future task instance pending for precomputation. Initially, assume that the task instance represented by curve 410 is executing and continues to do so through time slice .DELTA.t.sub.1. At the end of this slice, curve 410 has an illustrative magnitude 1.sub.1 which exceeds a magnitude then exhibited by future task instance curve 450. Consequently, the present task instance continues to execute. At the end of the next time slice, i.e. .DELTA.t.sub.2, curve 410 still exhibits a greater magnitude, i.e. here 1.sub.4, then does curve 450. Hence, current processing resources, here processing time, continue to be allocated to the presently executing task instance to further its execution and achieve current value thereby. Inasmuch as curve 410 begins to exhibit a downwardly concave shape starting in time slice .DELTA.t.sub.3 and continuing into time slice .DELTA.t.sub.4, the current task instance yields increasingly less incremental current value relative to that which can be had, discounted to the present, through precomputing the future task instance. Inasmuch as the incremental value provided by the currently executing task instance at the onset of each of time slices .DELTA.t.sub.3 and .DELTA.t.sub.4, i.e. magnitudes 1.sub.4 and 1.sub.3, respectively, still exceeds the EVC provided, on a discounted basis, by the future task instance at those times, processing resources continue to be allocated to the former task instance for the remainder of each of these intervals. However, at the onset of the next time slice, i.e. .DELTA.t.sub.5, the discounted EVC provided by the future task instance now exceeds the incremental value provided by current task instance, i.e. magnitude 1.sub.1. Consequently, since the future task instance will provide greater EVC, discounted to the present, processing resources, here processing time, are allocated to the former rather than the latter instance. Hence, the currently executing task instance is terminated in favor of precomputing the future task instance. Inasmuch as the discounted EVC provided through precomputation of the future task instance will continue, though time slice .DELTA.t.sub.6 to exceed the EVC of the current task instance (now suspended), precomputation of the future task instance will continue through this time slice as well. A similar graphical analysis can be made between any presently executing task instance and a future task(s) instance to determine whether precomputation of the latter should occur and when.

CLAIMS:

1. A method for use in a computer system, having a processor, for allocating and using currently available processing time comprising the steps of:

(a) defining a list having a plurality of future task instances, each of said instances having an incremental utility measure associated therewith resulting from allocation of computational resources to execution of said each future task instance;

(b) ascertaining, for each of said future task instances on said list, a corresponding time-discounted incremental utility measure therefor, said time-discounted utility measure being the incremental utility measure for said each future task discounted for time; and

(c) if the time-discounted incremental utility measure for one of said future task

instances exceeds an incremental utility measure of a currently executing task:

suspending the currently executing task prior to its completion so as to define a suspended task;

storing partial results of the suspended task for subsequent use; and

executing the one future task instance.

4. The method in claim 3 wherein the executing step comprises the steps of:

if the one future task instance completes during the remaining time in said each time interval, storing results of the one future task instance for subsequent use; and

if the one future task instance does not complete during the remaining time in said each time interval, storing partial results of the one future task instance obtained at a conclusion of said each time interval and suspending the one future task instance.

21. Apparatus for use in a computer system, having a processor, for allocating and using currently available processing time comprising:

the processor; and

a memory having executable instructions stored therein; and

wherein the processor, in response to the instructions stored therein:

(a) defines a list having a plurality of future task instances, each of said instances having an incremental utility measure associated therewith resulting from allocation of computational resources to execution of said each future task instance;

(b) ascertains, for each of said future task instances on said list, a corresponding time-discounted incremental utility measure therefor, said time-discounted utility measure being the incremental utility measure for said each future task discounted for time; and

(c) if the time-discounted incremental utility measure for one of said future task instances exceeds an incremental utility measure of a currently executing task:

suspends the currently executing task prior to its completion so as to define a suspended task;

stores partial results of the suspended task for subsequent use; and

executes the one future task instance.

23. The apparatus in claim 22 wherein the processor, in response to the stored instructions, executes said one future task instance during said each time interval to the extent of all remaining time in said each time interval.

24. The apparatus in claim 23 wherein the processor, in response to the stored instructions:

if the one future task instance completes during the remaining time in said each time interval, stores results of the one future task instance for subsequent use; and

if the one future task instance does not complete during the remaining time in said each time interval, stores partial results of the one future task instance obtained at a conclusion of said each time interval and suspends the one future task instance.

27. The apparatus in claim 26 wherein the processor, in response to the stored instructions and for each of said future task instances in the list:

determines a corresponding second associated rate of change in the expected value of computation, as a function of the corresponding ones of both of the second probability measures and the second utility measures; and

forms the time-discounted product as a product of a pre-defined numeric factor, the corresponding second associated rate of change and the second corresponding probability measure.

30. The apparatus in claim 27 wherein processor, in response to the stored instructions, executes said one future task instance during said each time interval to the extent of all remaining time in said each time interval.

31. The apparatus in claim 30 wherein the processor, in response to the stored instructions:

if the one future task instance completes during the remaining time in said each time interval, stores results of the one future task instance for subsequent use; and

if the one future task instance does not complete during the remaining time in said each time interval, stores partial results of the one future task instance obtained at a conclusion of said each time interval and suspends the one future task instance.

L44: Entry 7 of 11 File: USPT Nov 17, 1998

DOCUMENT-IDENTIFIER: US 5838968 A
TITLE: System and method for dynamic resource management across tasks in real-time operating systems

Brief Summary Text (15):
Finally programmers must manually determine how to balance resources between high priority tasks and low priority tasks. If a low priority task requires a large number of resources, but is not time critical, programmers have little choice but to hope that those resources are available when the low priority task runs. In static reservation systems a low priority task cannot pre-allocate all the resources it needs since that would make those resources unavailable for time critical tasks. Yet if a time critical task fails to deallocate its resources, the low priority task may never be able to execute since it cannot acquire the resources necessary to run.

Detailed Description Text (23):
Real time tasks execute under XOS 180 on the media engine chip, while resource manager 170 is responsible for creating and dynamically managing the resources available to tasks. Tasks have a task structure commonly known in the art as a task control block that contains the control information necessary for a task to run, be suspended and resumed.

CLAIMS:

4. A method of dynamic resource management on a data processing system including a processor, a plurality of system resources, and a plurality of tasks, the processor being coupled to the system resource and capable of supporting the tasks, the method comprising the steps of:

executing instructions on the processor to create a plurality of tasks capable of executing on the processor, each of the plurality having a priority;

creating a plurality of task resource utilization records for each of the plurality of tasks, each task resource utilization record of the plurality comprising quantities of the pluralities of system resources that each of the plurality of tasks qualitatively prefers to utilize while executing on the processor;

for each task resource utilization record of the plurality, assigning a run level to the task utilization record reflecting the associated task's ability to perform its work when allocated the resources according to each task resource utilization record; and

dynamically varying the quantity of the plurality of system resources that the plurality of tasks have allocated based on the availability of the plurality of system resources and the priorities of the plurality of tasks.

7. A method of dynamic resource management on a data processing system including a processor and a system resource coupled to the processor, the method comprising the steps of:

executing instructions on the processor to create a plurality of tasks capable of execution on the processor, each of the tasks having a priority;

creating a task resource utilization vector for each of the plurality of tasks, the task resource utilization vector including at least one task utilization resource record specifying a resource quantity request; and

dynamically varying the resource quantity requests for each of the plurality of tasks based on the availability of the system resource and the priorities of at least two of the plurality of tasks.

L44: Entry 8 of 11 File: USPT Jul 21, 1998

DOCUMENT-IDENTIFIER: US 5784616 A
TITLE: Apparatus and methods for optimally using available computer resources for task execution during idle-time for future task instances exhibiting incremental value with computation

Brief Summary Text (5):
Generally speaking, a single computer processor, while executing a given software program and under control of an associated operating system, executes a series of atomistic processes--which collectively implement the program. However, in any single sequential Von Neumann type processor, only one such process executes at any one time. A significant response time imbalance often exists between a rate at which the processor itself executes instructions for a given process and the substantially slower response time of a resource, e.g. a hard disk drive or other mass storage facility, utilized by that processor. Hence, when the process requires use of such a resource, the operating system issues a request for the resource and then ceases executing the process, i.e. the process is suspended and simply waits, until the resource has properly responded to the request. This wait time can become significant if the resource is busy, i.e. unavailable, at the time the request is issued. While the process itself could not advance its execution pending the availability of the resource, the computer processor itself is often not so constrained.

Brief Summary Text (24):
In accordance with the broad teachings of the present invention, presently available processing resources, here illustratively processing time, are allocated, within a computer system, to the task instance that will provide the largest "value" (or benefit), i.e. current fixed or incremental utility or future cost saving, to a current user of that system. Such an allocation can occur during idle-time intervals, in order to select a future task instance(s) most suited for precomputation thereduring, and/or alternatively during other processing intervals, such as during intervals of high or low activity, to determine whether enhanced current value will arise by prematurely suspending a currently executing task instance in favor of precomputing a future task instance.

Brief Summary Text (28):
In accordance with my specific inventive teachings, the desirability of continuing execution of a currently executing task instance vis-a-vis prematurely suspending the refinement of that instance in favor of precomputing a future task instance is assessed through use of discounted net expected value (NEV) exhibited by that instance. NEV is determined as a product, of the probability of a future task instance and its EVC flux, multiplied by a suitable time-discount factor. The currently executing task instance is terminated, i.e. suspended, in favor of precomputing a future task instance if, at the onset of a time slice, the latter instance exhibits a discounted NEV that exceeds the EVC flux then being provided by the former instance. Precomputation continues for the remainder of the time slice, with a re-evaluation of discounted NEV (including that of the suspended task instance) as against the EVC flux provided by the task instance then currently executing at the beginning of each successive slice, and so forth, in order to optimally allocate currently available processing resources to its best current use.

Detailed Description Text (48):
I have recognized that in certain situations, a future task instance(s), if precomputed, could provide greater expected value than the value provided by a task which is currently executing. In that regard, a portion of current task execution could be intentionally degraded or that task instance completely suspended, hence retarding or even halting execution of that task and freeing processing capacity, in favor of allocating that capacity to such a future task instance. Considering the net present value of results to be delivered by precomputing a future task instance necessitates time-discounting the net expected value of these results to the present. The discounted net expected value is then compared to losses in current value that would result from reducing or prematurely suspending the refinement of a current task instance. If the discounted value for any such future task instance(s) exceeds these losses, then a portion of present processing capacity is diverted from the presently executing task in favor of precomputing the future task instance(s).

Detailed Description Text (49):
An immediate loss of dedicating current resources, such as processing time, over a period of time is the product of that resource and average EVC flux over that period. The gains of allocating these resources to a future task instance is a function of the total amount of idle-time that will be presently available after a current task is degraded or prematurely suspended. If this idle time is sufficient, in duration, to fully process not only the present task instance but also all possible future task instances, then essentially nothing can be gained by transferring current resources to precomputing any future task instance. Hence, the expected value needs to be modified by a probability distribution, in terms of duration, taken over the available idle time.

Detailed Description Text (56):
Hence, in accordance with my inventive teachings, I conclude that current resources, such as processing time, should not be allocated to a future task if the EVC flux provided by the currently executing task instance exceeds the largest incremental utility measure of that future task instance, i.e., for example, the largest product of the probability of a future task instance and the EVC flux of that future task instance. Therefore, if for any time slice, the EVC flux for the currently executing task instance exceeds a largest incremental utility measure (i.e., the product of the EVC flux and the probability of the future task instance), then the affect on this product of discounting as well as the probability associated with idle time are both irrelevant and need not be considered, inasmuch as precomputation would be inferior to continuing the task instance that is currently executing during this slice. Alternatively, if precomputation is warranted, i.e. the discounted product exceeds the EVC flux for the currently executing task instance, then execution of this current task instance is suspended (which here includes its termination or merely retarding its execution, as appropriate) and the future task instance having the largest product of probability and EVC flux is precomputed to the fullest extent possible.

Detailed Description Text (58):
For future task instances having non-linearly varying EVC, the EVC flux of the presently executing task is compared to the discounted incremental utility measure, i.e., here discounted EVC flux-probability product, of the future tasks, with precomputation occurring only if the latter exceeds the former. This comparison could occur at either of two intervals: either at regular time slices, as described above, or, for each future task instance that is being precomputed, at the conclusion of that future task instance. In the latter situation, precomputation would continue only if the discounted EVC flux-probability product for the future task exceeded the EVC flux of the current task then suspended. As additional future task instances are encountered, their discounted EVC flux-probability products would be considered in an identical fashion. Likewise, as a future task instance is

precomputed, its discounted EVC flux-probability product is removed from further consideration.

Detailed Description Text (60):
To enhance understanding of the concept of allocating resources to precompute a future task instance at the cost of prematurely suspending a presently executing task instance, the reader should now consider FIG. 4.

Detailed Description Text (61):
This figure graphically depicts illustrative non-linearly varying .phi..times.p curve 410 for a currently executing task instance and illustrative linearly varying discounted .phi..times.p curve 450 associated with a future task instance pending for precomputation. Initially, assume that the task instance represented by curve 410 is executing and continues to do so through time slice .DELTA.t.sub.1. At the end of this slice, curve 410 has an illustrative magnitude 1.sub.1 which exceeds a magnitude then exhibited by future task instance curve 450. Consequently, the present task instance continues to execute. At the end of the next time slice, i.e. .DELTA.t.sub.2, curve 410 still exhibits a greater magnitude, i.e. here I.sub.4, then does curve 450. Hence, current processing resources, here processing time, continue to be allocated to the presently executing task instance to further its execution and achieve current value thereby. Inasmuch as curve 410 begins to exhibit a downwardly concave shape starting in time slice .DELTA.t.sub.3 and continuing into time slice .DELTA.t.sub.4, the current task instance yields increasingly less incremental current value relative to that which can be had, discounted to the present, through precomputing the future task instance. Inasmuch as the incremental value provided by the currently executing task instance at the onset of each of time slices .DELTA.t.sub.3 and .DELTA.t.sub.4, i.e. magnitudes 1.sub.4 and 1.sub.3, respectively, still exceeds the EVC provided, on a discounted basis, by the future task instance at those times, processing resources continue to be allocated to the former task instance for the remainder of each of these intervals. However, at the onset of the next time slice, i.e. .DELTA.t.sub.5, the discounted EVC provided by the future task instance now exceeds the incremental value provided by current task instance, i.e. magnitude 1.sub.1. Consequently, since the future task instance will provide greater EVC, discounted to the present, processing resources, here processing time, are allocated to the former rather than the latter instance. Hence, the currently executing task instance is terminated in favor of precomputing the future task instance. Inasmuch as the discounted EVC provided through precomputation of the future task instance will continue, though time slice .DELTA.t.sub.6 to exceed the EVC of the current task instance (now suspended), precomputation of the future task instance will continue through this time slice as well. A similar graphical analysis can be made between any presently executing task instance and a future task(s) instance to determine whether precomputation of the latter should occur and when.

CLAIMS:

30. The apparatus in claim 29 wherein the processor, in response to the stored instructions, executes said selected future task instance to the fullest extent possible during all remaining time in said first period.

L44: Entry 10 of 11 File: USPT Aug 5, 1986

DOCUMENT-IDENTIFIER: US 4604694 A
TITLE: Shared and exclusive access control

Abstract Text (1):
A method for controlling both shared and exclusive access for a resource in a multiprocessor system wherein a first-in/first-out queue is formed for tasks suspended while awaiting access and wherein access to the resource provides that control of access required for manipulation of the first-in/first-out queue which is not provided by the atomic nature of compare (double) and swap. Each member of the queue has indicators of the access it requested and of the next most recently enqueued member which has a corresponding indicator. A lockword is established having two parts, a lock flag indicating the status of the resource, whether available, under shared ownership or under exclusive ownership and a lock pointer pointing to the most recently enqueued task. In requesting or releasing access, an initial guess is made as to the value of the lockword and a projected lockword is calculated based on the guess. Then an atomic reference is made to the lockword during which no other multiprocessor has access to the lockword. During the atomic reference, the lockword is compared to the guess of the lockword and if the guess is correct, the lockword is replaced by the projected lockword which rearranges the queue for the requesting or releasing task. If the guess was incorrect, the value of the lockword is used to calculate another projected lockword. If another task can affect the next tasks to gain access, the process with the atomic reference is repeated until no intervening changes occur between atomic references.

Brief Summary Text (9):
In an IBM System/370, designed for a multi-tasking environment, there is a test and set instruction which can fetch a word from memory, test for a specific bit and return a modified word to the memory, all during one operation in which all other tasks or processors are barred from accessing that particular word in memory. The fetch and return-store forms an atomic unit or atomic reference which, once begun, cannot be interrupted by or interleaved with any other CPU in a multi-processor. The test and set instruction can therefore be used to test a lockword and to set it for ownership. The set of operations is described in Table 1 in which one bit of the byte LOCKWORD is tested for zero, indicating availability of the lockword. LOCKWORD is immediately rewritten with this bit set to a "1". The result of this testing is retained and used in the next step by a conditional branch BC. If the testing was not successful, i.e, the lockword was owned by another task or processor, execution branches back to retry, the test and set operation. When the lockword is available and ownership of the lockword is established, a series of operations are performed in which the queue is manipulated by this requesting task or processor. While this manipulation is proceeding, no other task can manipulate the queue because this task owns the lockword. When the manipulation has been completed, a final instruction rewrites the lockword to indicate that it is once more available. LOCKWORD is set to zero, indicating that the queue is once more available to other requesting tasks or processors.

Brief Summary Text (12):
The enhanced spin locks are, in turn, used to control the manipulation of queues for which tasks, but not processors, are suspended until the required availability. Requests which allow concurrent ownership to others are called "shared" requests.

Requests which allow no other component ownership are called "exclusive" requests. Requests which cause the task to be suspended without suspension of a processor are called "task locks".

Brief Summary Text (19):
Because the suspension and resumption of a task cannot itself be suspended and resumed as a task, the lock which controls its queues must necessarily be a spin lock. If this spin lock is not the same lock used to control access to the controls used to provide shared access, another level of locking may be introduced when a task must be suspended or resumed for the lack of availability or the reappearance of availability of a resource. These three levels are an improvement over the four levels required with test and set. However, the three levels still introduce system complexity with shared tasks. Furthermore, they contribute to unwanted system complexity and slow its operation.

Brief Summary Text (21):
Accordingly, an object of this invention is to provide task locking with a minimum level of operations between the instruction set and the application task locking.

Detailed Description Text (2):
The architecture of the task locking according to this invention will be described with reference to the block diagram of FIG. 3. If one or more tasks has requested access to a resource but their requests cannot be honored, then the request is put into a queue. The first queue request can be rejected because the task currently executing on the resource is a task requiring exclusive access to that resource or the requesting task can itself be requesting exclusive access when the resource is already owned either exclusively or shared. If the resource is not currently busy, a request is immediately honored and no queue is formed. In the absence of a queue, if one or more tasks currently have shared access to the resource, than an additional request for shared access is immediately honored and there is no reason to form a queue. The queue will have the architecture of a first-in/first-out queue. That is, a request for access to the resource is honored for the oldest or least recently submitted request before a more recent request is honored. This means that if an older request is a request for exclusive access, a more recent request for shared access will be denied, even if the resource is currently being used with shared access. If the resource is currently in use for shared access, then the top of the queue or the least recently enqueued task will necessarily be a request for exclusive access. Previously submitted requests for shared access would have been honored and the associated task removed from the queue. However, requests for shared access, less senior than an enqueued exclusive request, may be in the queue. The queue is formed of a series of task deferral control blocks (TDCB) arranged at arbitrary locations in a memory 22 of the multi-processor system. It should be noted that the resource may also be a part of the memory 22. A separate task deferral control block is set up for each task that has been suspended because a request for access to the resource has been denied. Each task deferral control block contains a variety of information. It must contain all information required to resume the suspended task, such as a pointer to the task or a control program event control block to be posted. It must also contain an indication specifying whether shared or exclusive access SH/EXCL was requested for this task. Of course this indication can be omitted if only one form of access is permitted. Finally, for the purposes of this invention, it contains a pointer NCB to the next most recently enqueued task deferral control block, if any. The NCB of the top, least recently queued elements of the queue is set to zero.

Detailed Description Text (10):
If the resource is not now available for exclusive access, execution of the request reaches point 32 and preparations are made for putting the current or requesting task into a queue. A task deferral control block is prepared and the contents of this block are set up for the current task along with an indication that it is a request for exclusive access. Preparations are made to put the current task into

the queue, which previously may not have existed. At this point 34, the best current prediction for the value of the lockword is the value OLD obtained in the compare double and swap 31. The value of LOCKFLAG would not change if the current task is put in the queue so NEWFLAG is replaced by OLDFLAG. However, the updated LOCKTPTR would point to the current task which upon enqueuing would be the most recently enqueued task. The NCB of the task deferral control block for the current task would point to the task deferral control block pointed to by the previous LOCKTPTR, at that time the most recently enqueued task. Accordingly, the value of NEWTPTR is replaced with the identification, normally the address, of the task deferral control block of the current requesting task and the next block pointer NCB in the current task deferral control block is replaced with the value of OLDTPTR. Then another atomic reference 36 in a compare double and swap is performed. If the lockword remains as it previously was so that OLD=LOCK, then LOCK is replaced with NEW and the current task is placed in the queue with the pointers corrected for the correspondingly rearranged queue. The execution of the current task is suspended as it awaits access to the resource and the task is enqueued. If, however, since the last compare double and swap 31, some other task has rearranged the queue by modifying the lockword, then OLD does not match LOCK and the current value of LOCK is placed into OLD.

Detailed Description Text (11):
At this point 38, a decision is made as to whether the change in state is due to the resource becoming available or whether the resource is again unavailable but the queue has been rearranged. If OLD=(0, 0), then the best guess is that the resource is now available so that it is no longer necessary to form a queue. The space for the task deferral control block for the current task is de-allocated if necessary, and execution returns to the start 30. However, if OLD.noteq.(0, 0), then a change has been made to the queue but the resource is not available. Execution then returns to point 34 with the value of OLD having been obtained from the compare double and swap of the recent atomic reference 36 rather than first atomic reference 31.

Detailed Description Text (16):
Then a new atomic reference 50 is made using compare double and swap. If the lockword has not since the last atomic reference, OLD matches LOCK and the lockword is updated by the predicted new lockword by replacing LOCK with NEW so that the current task is properly enqueued. The current task is then suspended awaiting availability of the resource for which it has been enqueued.

Detailed Description Text (29):
At the point 90 where NEWFLAG has been set to its proper value, a test is made as to whether a queue will exist after the intended re-arrangement of the queue. If BAKPTR.noteq.0, a queue exists. In this case 92, the next control block pointer NCB of the task deferral control block pointed to by BAKPTR is set to 0, indicating that this task will become the least recently enqueued task in the queue and LOCKFLAG is set to NEWFLAG in an MP-consistent operation. At this point 94, the queue has been rearranged and all that remains to be done is a resumption of the suspended tasks, disposition of the no longer used task deferral control blocks and a release of access by abnormally terminated tasks in block 96.

Detailed Description Text (30):
If, however, BAKPTR=0 execution reaches point 98 indicating that no queue will be further necessary. In this case NEWTPTR is set to 0 to indicate the intended absence of a queue following atomic reference 100 with a compare double and swap. A successful test for OLD=LOCK means that no rearrangement of the queue has occurred since the last atomic reference. In this case LOCK is set to NEW and execution reaches point 94 for the abnormal termination clean-up, disposition of the task deferral control blocks and resumption of suspended tasks. If, however, the test was unsuccessful, OLD is replaced by LOCK and execution returns to point 84 to once again find the status of the queue.

Detailed Description Text (35):
It is seen that the supervisor task lock can be accomplished without spin-locks except as may be required when suspending or resuming task execution. Of course, supervisor spins are possible with compare double and swap. The advantages of the absence of spinning within the described methods are the elimination of the associated cost in performance, the ability to exploit this form of locking directly from multi-task, multi-processor applications, and a reduction in the concern for recovery from the CPU failure in which a spin-lock is held by the failing CPU.